APPENDIX

F

BEHIND THE SCENES WITH DEBUG

This appendix uses DEBUG, a DOS and Windows 9x editor, to view and manipulate the components of the file system on floppy disks and hard drives, including the FAT, the directories, the boot record, and files. Many good data recovery software products are available today that can recover data automatically, and you don't have to understand exactly what is going on behind the scenes to use them. If you do want a more hands-on sense of the process, however, this appendix describes how to interpret what's written on the disk, how a disk becomes damaged, and a manual method of data recovery that gives you the most control over what is happening. This approach is not always the best or fastest way to recover lost data, but it is certainly the most reliable.

The reasons for using the not-too-user-friendly DOS and Windows 9x DEBUG utility are (1) everyone who has DOS or Windows 9x on his or her computer has DEBUG, and (2) with DEBUG, you can see and learn at the "grass roots" level what is written on a disk. Learning about data recovery using DEBUG provides the strong technical insight that you need to use more user-friendly utility software, such as Nuts & Bolts, Norton, and Lost & Found, and to be confident that you understand how it works. The better you understand how data is constructed on the disk and exactly what problems can arise, the better your chances of recovering lost or damaged data.

INTRODUCING THE DEBUG UTILITY

The DOS and Windows 9x DEBUG utility is a tool that displays for you the hex values of the contents of memory or anything written to a floppy disk or hard drive including the FAT, directories, boot record, data, and so on. This information appears exactly as it is written to the drive.

Using DEBUG is not difficult if you grasp one concept: DEBUG is an editor. It is just like any other editor except that it displays and works with hex values rather than filenames or numbers or other familiar terminology. For instance, when you use Notepad to open a file and edit it, these events occur: The file is "opened" by copying it from the hard drive or disk into memory. Notepad, an ASCII editor, displays the contents of memory to you in ASCII form and allows you to edit what is in memory. When you "save" the file, Notepad copies the contents of memory back to the floppy disk or hard drive. DEBUG performs similar functions only in hexadecimal, but the commands are written differently. DEBUG works beyond the concept of a file, looking directly at the bits stored in sectors on the hard drive, converting them to hex before displaying them. DEBUG "opens" a sector by copying the sector from the disk or hard drive into memory and then displaying the contents of memory. You must tell DEBUG which sector to read from the drive and which memory addresses to copy the sector into. You can display and edit the contents of these memory addresses and then command DEBUG to copy the contents of memory back to the drive, which is similar to the "save" concept in Notepad.

The advantages of using DEBUG are that (1) you circumvent the limitation placed on you by the OS and software applications, which only let you view files; with DEBUG, you can look at any sector on the hard drive or disk regardless of its function, and (2) by viewing critical sectors that are used to organize the data on a hard drive, you gain knowledge invaluable to you when using Nuts & Bolts or Norton Utilities to recover the data on a damaged hard drive or floppy disk.

Table F-1 compares the functions of Notepad and DEBUG to further help you become comfortable with this interesting tool.

Using DEBUG

To use DEBUG you need to understand the hexadecimal number system and how memory is addressed in microcomputers by using the segment address followed by a colon and the offset. Appendix D covers these topics. If you have not already done so, read that appendix before proceeding. Follow these steps:

To begin, shut down, power off your computer, then power on your computer with only the essential software loaded, so that memory is relatively free and easily identified as unused. This makes it easier to find some unused part of memory to use as a "scratch pad" as you work. The easiest way to boot with only essential software loaded is to boot from a floppy disk.

When you boot from a floppy disk, you will be at an A prompt. Go to the C prompt by typing **C:**

Table F-1 Notepad and DEBUG are both editors

Notepad Function	DEBUG Function	Description of Functions
Open a file. (Use the **File**, **Open** command.)	Copy a sector from disk to specified memory addresses. (Use the **L** or **Load** command.)	Gives you a "snapshot" of data written on a disk
Display contents of file now stored in memory. (Notepad does this automatically.)	Display the contents of the memory addresses above. (Use the **D** or **Dump** command.)	Displays contents of memory
Edit a file.	Change the contents of memory. (Use the **E** or **Enter** command.)	Changes the data displayed that is stored in memory, but does not change the data on the disk
Save or close a file. (Use the **File**, **Save** or **File**, **Close** command.)	Write the contents of memory back to the disk. (Use the **W** or **Write** command.)	Writes data from memory to disk
Exit the Notepad editor. (Use the **File**, **Exit**, command.)	Exit the DEBUG editor. (Use the **Q** or **Quit** command.)	Exits the editor

Execute DEBUG. The DEBUG program should be stored in the \DOS directory of your hard drive for DOS, and in the \WINDOWS\COMMAND directory for Windows 9x. (Later, to exit DEBUG, type **Q** to quit.) The paths should be:

```
\DOS\DEBUG
```

or

```
WINDOWS\COMMAND\DEBUG
```

Next, look for an area of memory where you can store whatever it is you will be examining. Suppose you plan to examine the boot record of the floppy disk that you booted from. You know that the boot record is one sector long, which is 512 bytes. Therefore, you need 512 bytes of unused memory to store the boot record. Use the DUMP command to search for an unused area of memory. Near the top of conventional memory is a good place to look for some empty area to use as a "scratch pad." Try the memory location 5000:0. If you find data there, keep looking until you find an empty area of memory. Enter this dump command:

```
-d5000:0
```

You should now see 128 bytes of memory beginning with the location 5000:0. When you press **d** followed by no parameters, DEBUG gives you the contents of the next 128 bytes of memory. In Figure F-1, you see that memory is clean (i.e., filled with zeroes and containing no data) from 5000:0000 through 5000:00FF. That equals 256 bytes. Because a sector on a 3½-inch high-density disk is 512 bytes, issue the DUMP command twice more to make sure that a total of 512 bytes of memory is unused:

```
-d
-d
```

You now know that copying the boot record into this area of memory will not overwrite something important. If your dump displays data in memory at this location, try some other area until you find 512 free bytes.

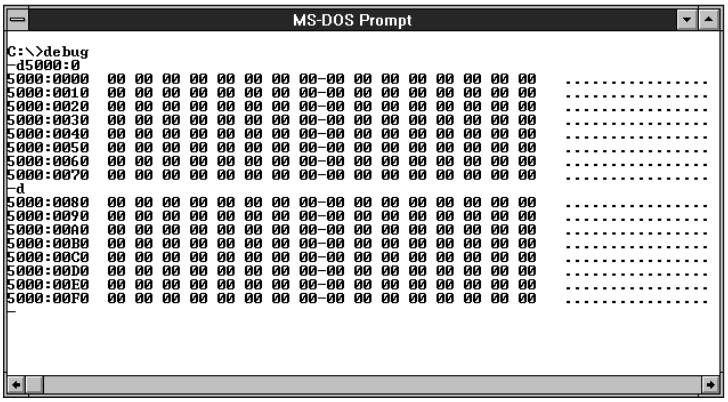

Figure F-1 Empty area of memory from 5000:0 through 5000:00FF (two dump commands)

Next we examine the boot record of a floppy disk. You could examine the boot record of a logical drive on a hard drive using the same basic approach.

The Boot Record

This section uses DEBUG to examine a good boot record of a floppy disk and then repair a corrupted one. Recall from Chapter 5 that a 3½-inch floppy disk has 512 bytes in each allocation unit or cluster, which means there is one sector in each cluster. Also, there are 2,847 clusters or sectors on the disk.

To examine the boot record, first execute DEBUG and then load the boot record into an unused area of memory. Use this command:

```
-L5000:0 0 0 1
```

The command line is interpreted as follows:

L5000:0	Load into the area of memory beginning at location 5000:0.
0	Load from the disk in drive A (drive B is 1, and drive C is 2).
0	Begin with sector 0.
1	Load 1 sector.

Look at the contents of memory using the DUMP command:

```
-d5000:0
```

Follow this with three more **d** commands to see the entire 512-byte sector. Figure F-2 shows the first 128 bytes of the boot record of a 3½-inch disk formatted with DOS. Figure F-3 shows the beginning of the boot record for the same size disk formatted with Windows 95, for comparison. On the diskette, when more than one byte is used to express a number, the least significant byte is written first, which is the reverse of the way we normally write a value. Values

shown on the right side of these two figures are written with the most significant byte first, followed by the least significant byte. For example, the first value at the top of Figure F-2, bytes per sector, is written on the diskette as 00 02. To write the hex value, reverse the bytes, which gives 0200h as the number of bytes per sector. Note that most of the data on the right side of the screen cannot be interpreted by the editor. This section is converting the contents of memory in hex numbers to their ASCII representation, as described in Appendix C; most of this data is not ASCII code and, therefore, makes no sense when converted to ASCII.

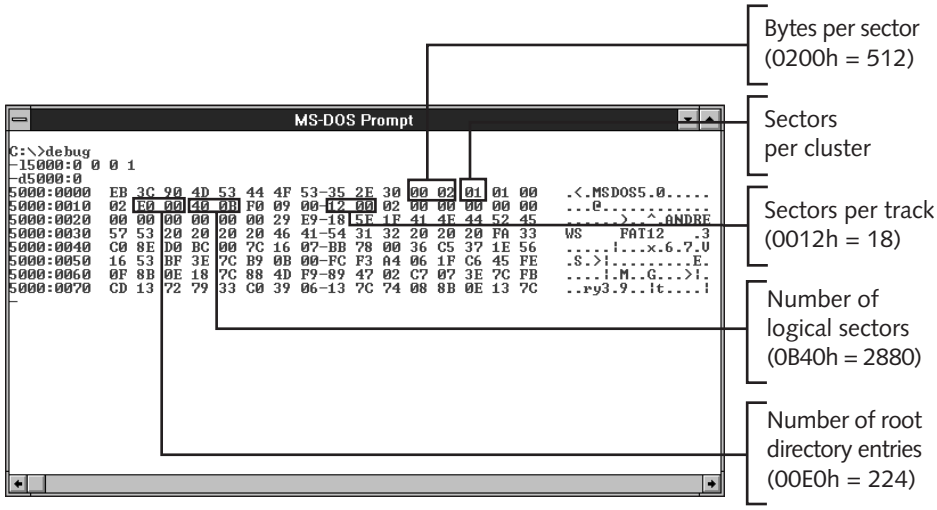

Figure F-2 The first 128 bytes of the boot record of a 3½-inch floppy disk formatted with DOS

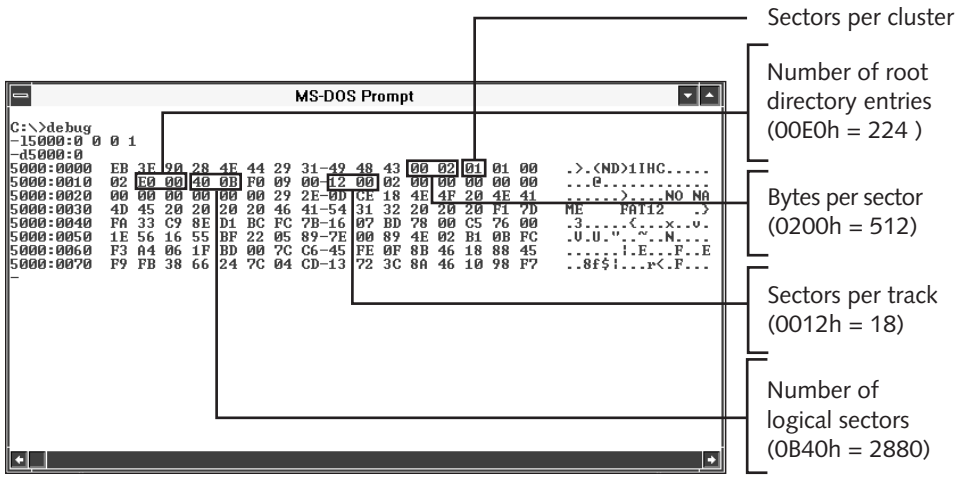

Figure F-3 The first 128 bytes of the boot record of a 3½-inch floppy disk formatted with Windows 95

The figure labels highlight some interesting entries at the beginning of the record. Table 6-4 of Chapter 6 shows the complete record layout for the boot record. The medium descriptor byte tells DOS what type of disk this is. The values of this descriptor byte are shown in Table 6-5 of Chapter 6.

The program code that loads DOS starts at location 5000:0200. In the last 128 bytes of the record shown in Figure F-4, you can see that the message that prints if this program does not find the hidden files that it needs to load DOS is as follows:

```
Non-System disk or disk error..Replace and press any key
when ready...
```

At the bottom of the record are the names of the files that it is searching for, in this case IO.SYS and MSDOS.SYS. Continue to use the DUMP command until you can see all of these entries. Figure F-5 shows a similar presentation of this message for a Windows 95 disk.

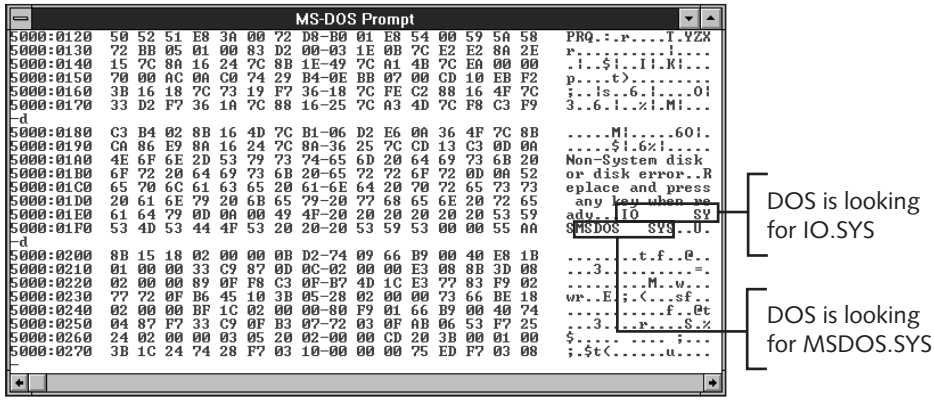

Figure F-4 "Non-system disk" message for a DOS disk

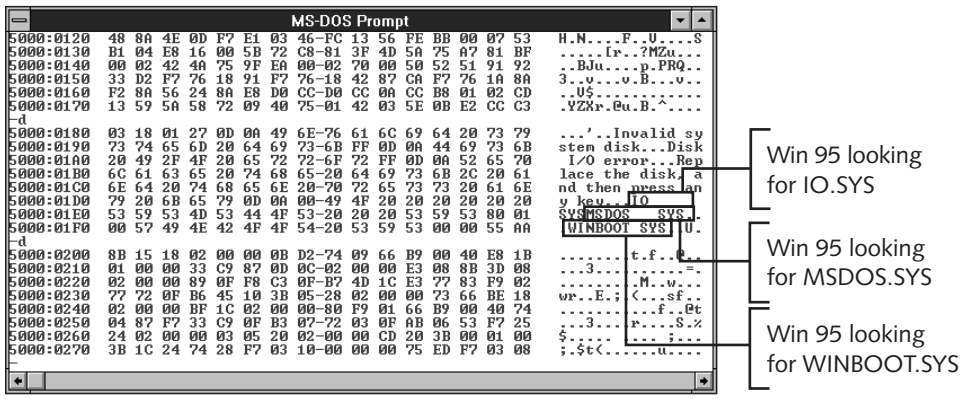

Figure F-5 "Invalid system disk" message for a Windows 95 disk

To repair a damaged boot record using DEBUG, follow this procedure:

- Find or make a disk of the same size and density as the damaged disk.
- Using the procedure above, load this good boot record into memory.
- Copy the good boot record from memory to the damaged disk, using this command:

```
-W5000:0 0 0 1
```

The command line is interpreted as:

W5000:0	Write beginning with the data located at memory address 5000:0.
0	Write to drive A (B = 1 and C = 2).
0	Write to sector 0.
1	Fill one sector.

The File Allocation Table (FAT)

After the boot record, the next file system component on a disk or drive is the FAT. This section explains how a 16-bit FAT on a hard drive actually appears by viewing the FAT with DEBUG. First look at Figure F-6, where a FAT is shown that contains two files. Figure F-7 shows the same information as a screenshot of the beginning of a 16-bit FAT for a hard drive created using DEBUG. Compare the two figures as you read this section and notice precisely what the FAT looks like to the OS. The memory dump displayed in Figure F-7 was created using the DEBUG utility with these commands:

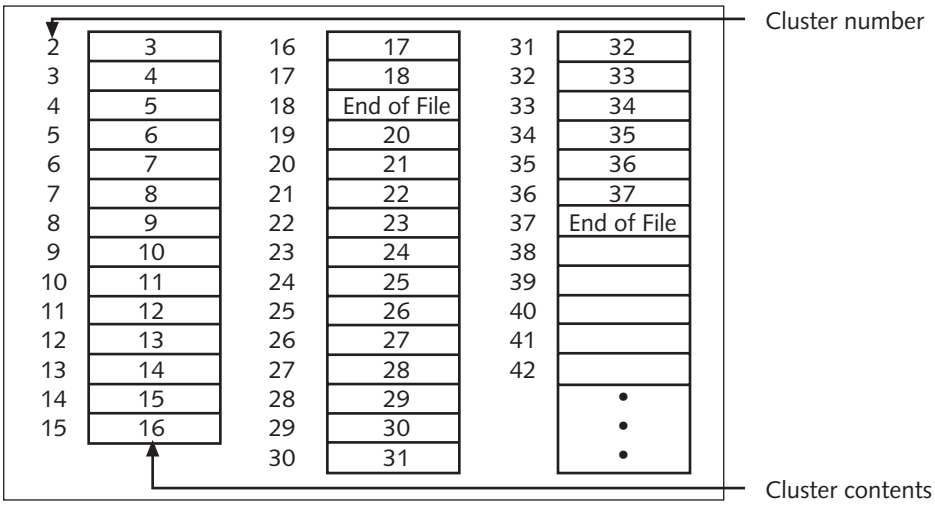

Figure F-6 FAT showing two files

Command	Description
C:\>DEBUG	Execute DOS DEBUG.
–L9000:0 2 1 1	From drive 2 (drive C), beginning with sector 1 and reading 1 sector, load contents into memory addresses beginning with 9000:0 (Or substitute another area of memory if this area is used).
–D9000:0	Dump to screen the contents of memory beginning at 9000:0.

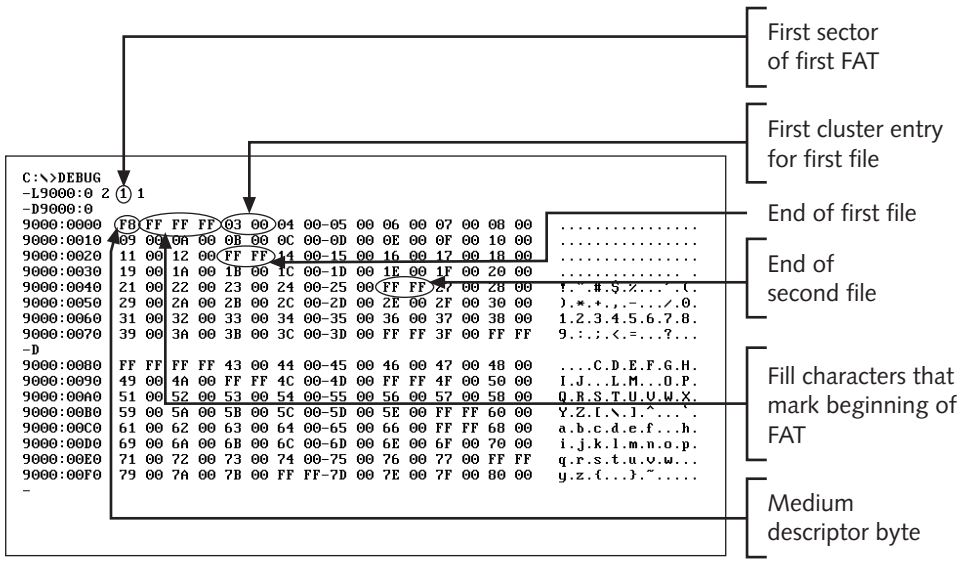

Figure F-7 Beginning of a 16-bit FAT on a hard drive

Here is what the data in the dump means. The first byte in Figure F-7 is F8, the byte that identifies what medium is used—in this case, a hard drive. The next three bytes are FF FF FF, which indicate the beginning of the FAT. Each of the following entries in the FAT uses two bytes. The next few bytes describe the location of the first file on the disk, which begins at cluster 2. The entry for cluster 2 is 03 00. Because the contents of memory are written from right to left, you must reverse the order of the two bytes to read the cluster number. Thus, you read the 03 00 entry as 00 03, which means that the next cluster used by this file is cluster 3. The file occupies 17 clusters, ending at cluster 12h with FF FF, which indicates the end of the file. The clusters used, then, are numbered 2 through 12 (hex) or 2 through 18 (decimal). This file occupies 2,048 bytes/cluster × 17 clusters, or 34,816 bytes of disk space. In a later section, we will look at the file's entries in the root directory.

In Figure F-7 the next file begins at cluster 13h and continues through cluster 25h. A quick count shows that this second file occupies 19 clusters, which equal 2,048 bytes/cluster × 19 clusters, or 38,912 bytes of disk space.

The FAT begins at sector 1. This particular hard drive's logical geometry is 1,024 cylinders, 17 sectors per track, and 12 heads. (To find out how many cylinders, sectors per track, and heads your hard drive has, so that you can make similar calculations, see your CMOS setup.)

How many individual sectors and how many clusters are there, and how long is the FAT in the example discussed here? You calculate as follows:

1,024 tracks × 17 sectors/track × 12 heads = 208,896 sectors
208,896 sectors / 4 sectors per cluster = 52,224 clusters

This hard drive can accommodate only 52,224 files and subdirectories, which equals the total number of clusters. Each cluster has a 2-byte entry in the FAT, making the size of the FAT as follows:

52,224 FAT entries × 2 bytes/FAT entry = 104,448 bytes
104,448 bytes / 512 bytes per sector = 204 sectors

A hard drive keeps two copies of the FAT, just as disks do, so you can expect the second copy of the FAT to begin at sector 205. Use DEBUG to verify your calculation for your machine. To use the Load command in DEBUG, you must convert 205 to hex, which is CDh. You can use these commands to display the second copy of the FAT, shown in Figure F-8.

```
C:\>DEBUG-L9000:0 2 CD 1
```

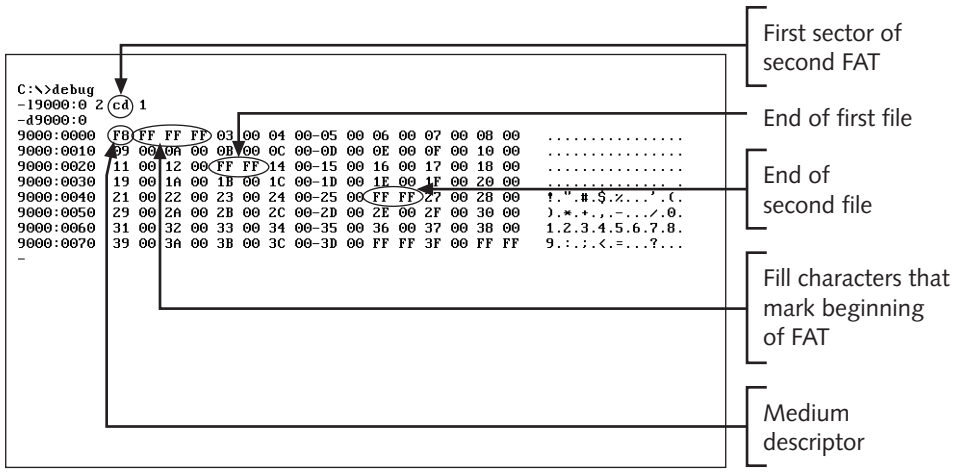

Figure F-8 Second copy of FAT

Execute DEBUG. Load into memory beginning with memory address 9000:0 the contents of disk 2 (drive C) beginning with sector CDh and continuing for 1 sector.

```
-D9000:0
```

Dump the contents of memory beginning with 9000:0.

How a Floppy Disk FAT is Written and Used

This section is designed to help you understand how the FAT is written to a floppy disk and how it is used. This will allow you to better use Norton Disk Editor or similar editors to recover data from a disk and ultimately from a hard drive. The discussion below uses DEBUG

as a tool to demonstrate what these editors are really doing. The discussion first takes a look at some healthy FATs.

Figure F-9 shows a FAT of a Windows 98 freshly formatted disk that does not contain system files; this is the emptiest FAT possible. The same results can be obtained using DOS to format the disk.

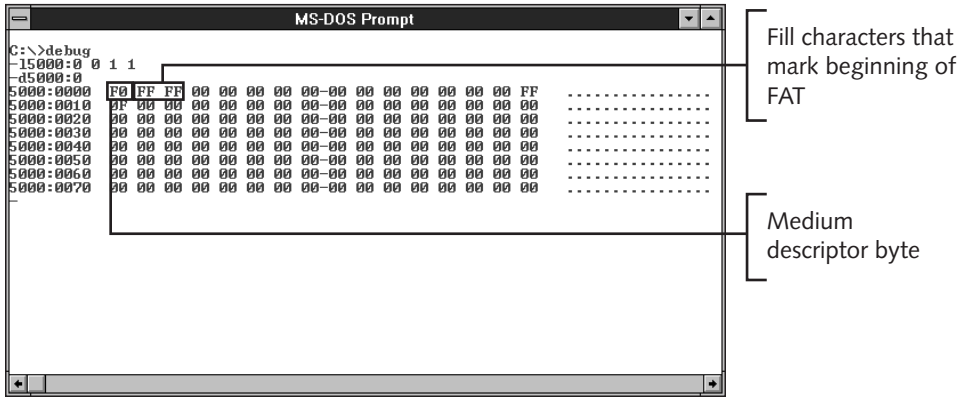

Figure F-9 Empty 12-bit FAT on a floppy disk

The FAT in Figure F-9 can be produced using this procedure:

1. Format a 3½-inch high-density floppy disk using Windows 9x Explorer.
2. Load the first copy of the FAT into an empty area of memory, using the following command substituting a different area of memory as necessary:

```
C:\>\WINDOWS\COMMAND\DEBUG
-L5000:0 0 1 1
```

An explanation of the command line is as follows:

L5000:0	Load data into memory addresses beginning at 5000:0.
0	Load from the disk in Drive A (A = 0, B = 1, C = 2).
1	Begin the load with sector 1.
1	Load one sector.

Remember that the boot record is located at sector 0, and the FAT begins at sector 1. Looking at Figure F-9, you see that the first byte is the medium descriptor byte, which tells DOS what kind of disk is being used. The next two bytes are called fill characters and are always FF FF for a 12-bit FAT, and FF FF FF for a 16-bit FAT. The rest of the FAT is empty.

Next WordPad is used to create a file on this disk and save the file as a DOS text file using the choices given in the WordPad Save As command. Name the file MYFILE.TXT and put in it the five characters "HAPPY". Figure F-10 shows the result of the DOS DIR command and the newly dumped FAT. Note first in Figure F-10 the difference between the number of free bytes on the disk and the number of free bytes before you created the five-character file:

```
1,457,664 bytes – 1,457,152 bytes = 512 bytes
```

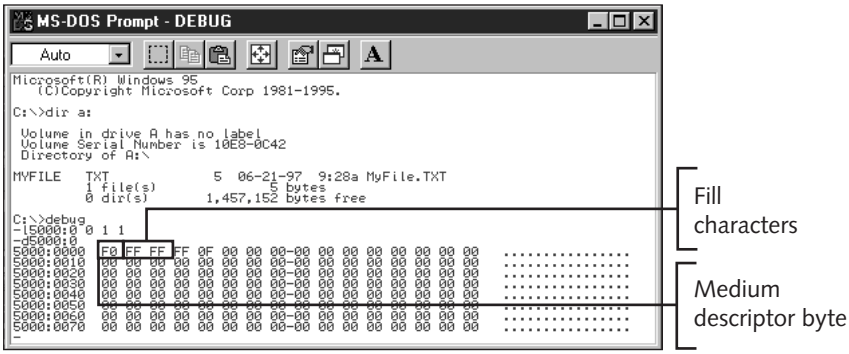

Figure F-10 Directory of disk showing MYFILE.TXT followed by a DEBUG dump displaying FAT with five-character file

F

The five-character or five-byte file took up 512 bytes of disk space! Disk space is wasted because one file allocation unit or one cluster on this disk equals 512 bytes. Compare the FAT in Figure F-10 with the one in Figure F-9. Note that two bytes are now altered, and read

```
FF 0F
```

Before you begin interpreting these two bytes, four facts need to be pointed out:

- All floppy disks have 12-bit FATs, and each entry in the table is 12 bits long. These 12 bits equal 1.5 bytes.
- Entries in a FAT are written in reverse order. In other words, if you label two bytes 1 and 2 in the FAT, you read the bytes as 2 and 1 when reading the FAT entry. In Figure F-10, you read the second byte first, so the bytes are read 0F FF.
- The end of a file in the FAT is written as all 12 bits filled with 1s. This record (1111 1111 1111) is converted to FFF in hex, which is what you see in the FAT entry.
- Clusters are numbered in the FAT beginning with cluster 2. Cluster 2 in a FAT entry is really not physically the second cluster on the disk, because the boot record, the two copies of the FAT, and the directory table all come before the data files. Windows 9x and DOS both number the clusters allocated for data beginning with 2.

Now back to Figure F-10. You know that your file only takes up one cluster, or one sector, and you would expect that to mean only one entry in the FAT. You interpret the first 2 bytes in reverse order as 0F FF. One entry is 12 bits, or 1.5 bytes, long, and the end of a file is identified with FFF. You see then that this file has the one entry, FFF.

Next, you copy to the disk a second file named Long Word File.doc that uses a long file name and contains 4,608 characters, or bytes. Examining the result of the CHKDSK command in Figure F-11, you see the short DOS version of the filename and see that the number of clusters, or sectors, required by this second file is:

```
4608 bytes/512 bytes per cluster = 9 clusters
```

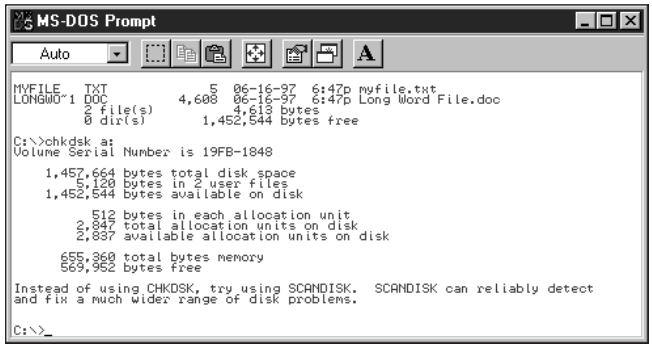

Figure F-11 Disk with two files

Since you can't allocate partial clusters, if the number of clusters had been a fractional number you would have rounded up to the nearest whole cluster; this file, however, requires exactly nine clusters. Adding one cluster for your original file, you get a total of 10 clusters used on the disk. Figure F-12 shows a dump of the FAT with the two files stored on the disk. Since the FAT entries are 1.5 bytes long, you will first divide these bytes into groups of 3 bytes apiece. Each group of 3 bytes contains two FAT entries and can be interpreted as follows:

```
AB CD EF = DAB EFC
```

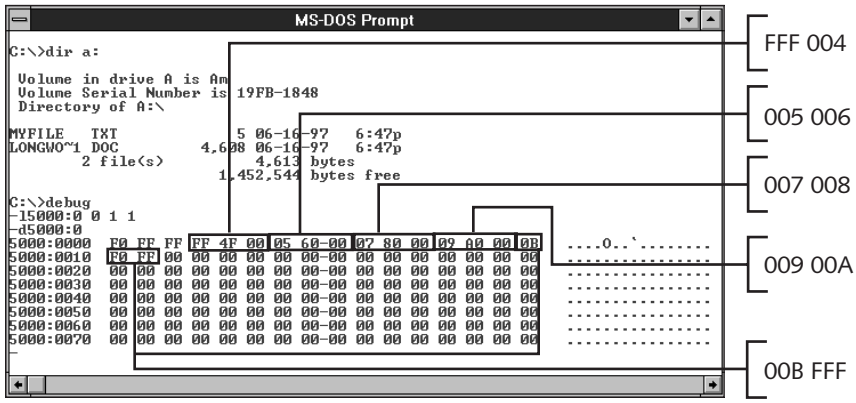

Figure F-12 Dump of the FAT showing two files

AB CD EF represents two FAT entries of 1½ bytes each. The first entry is AB and half of the next entry, CD. However, since the entries are written reversed, the part of the CD entry that belongs to the first FAT entry is the D, not the C. Also, the first entry is written going from byte 2 (CD) to byte 1 (AB) so that the entry is DAB. Similarly, the second entry begins with the EF and ends with C.

For example, say the first 3 bytes are FF 4F 00. We reverse these bytes and read them as FFF 004. The first entry marks the end of the first file, which is one cluster long. The second entry points to cluster 4. Remember that DOS begins with 2 when counting clusters. These two

entries contain the pointers for cluster 2 and cluster 3. Cluster 3, the first cluster of Long Word File.doc, tells DOS to continue on to cluster 4 to find more of Long Word File.doc. Reading from Figure F-12, you see that cluster 4 points to 5, 5 points to 6, 6 points to 7, 7 points to 8, 8 points to 9, 9 points to A, A points to B, and B marks the end of the file with FFF.

Restoring a Damaged FAT Using DEBUG

With a great deal of patience, you could (1) read a disk sector by sector marking the beginning and ending of a file, (2) calculate the FAT entries, and (3) reconstruct a damaged FAT by editing it using DEBUG. To do this (besides needing tons of patience), you only need one more DEBUG command, the E or ENTER command, which allows you to change the contents of memory. You can load a damaged FAT into memory, make the necessary changes with the ENTER command, and write the corrected FAT back to the disk.

For example, suppose you decide to limit the file Long Word File.doc to three clusters. You want to put the FFF at the FAT table entry 5 so that only clusters 3, 4, and 5 are allocated to the file. The command is:

```
-E5000:0007
```

DEBUG responds with 5000:0007 60.

Whatever you now type replaces the 60 currently at this location. You respond with 5000:0007 60.F0 (typing only the F0).

Pressing the Spacebar, you can continue to enter new data that replaces whatever is stored in memory from this location forward. Therefore, instead of only typing F0, you can correct both bytes like this:

```
5000:0007 60.F0
5000:0008 00.FF
```

In the first line, everything up to the first period was displayed. You typed F0 and one space. DEBUG responded with the contents of the next byte, which was 00 followed by a period. You then typed FF and pressed **Enter**, which changed the second byte and terminated the command.

In the last step, you write the altered FAT to disk with this command:

```
-W5000:0 1 1 1
```

The command causes the one sector to be written to sector 1 of the A drive disk.

There is only one problem. Windows 9x or DOS is expecting a file that is 4,608 bytes long, but you have given it only three clusters, which is not enough. There is one last thing to do: you must alter the root directory to change the size of the file, so that the OS will expect to find three clusters when reading the file. Editing a directory table is coming up.

Second Copy of the FAT

Recall that Windows 9x and DOS keep a second copy of the FAT immediately following the first copy. Without utility software, this second copy is not available to you if the first copy

is damaged. You can look at the second copy of the FAT on a 3½-inch high-density disk with this DEBUG command:

```
-L5000:0 0 A 1
```

The second FAT begins with sector A in hex or 10 in decimal. The first FAT began at sector 1. A quick calculation tells you that the first FAT requires 9 sectors or 512 bytes/sector × 9 sectors = 4,608 bytes. You see this must be so because this disk has 80 tracks/head × 18 sectors/track × 2 heads/disk, or 2,880 sectors (see Table 5-1 of Chapter 5). Because one sector equals one cluster, the FAT has 2,880 entries. Since each entry is 1.5 bytes long, the FAT contains 4,320 bytes, which is equal to 8.44 sectors. Hence, 9 sectors for one FAT are all accounted for. If you're reconstructing a FAT using DEBUG, don't forget to alter the second FAT as well as the first.

Even more important, maybe the second FAT isn't damaged. Load it into memory and see. If it isn't damaged, write it to the disk into the first FAT's location, and your disk is restored.

Examining and Repairing Directories

Recall that the root directory is created when a floppy disk or logical drive is first formatted. Later, subdirectories are created under the root which have a similar structure. This section describes how to examine and repair the root directory and subdirectories using DEBUG.

The Root Directory

The root directory is written on a hard drive immediately after the second copy of the FAT. For the hard drive FAT shown in Figure F-8, the second copy of the FAT ends at sector 408, and the root directory begins at sector 409, which is 199h. The dump of the beginning of a root directory is shown in Figure F-13. You saw in Figure F-7 that the first file on this hard drive occupies 34,816 bytes of disk space and begins at cluster 2. The second file on the hard drive begins at cluster 13h and occupies 38,912 bytes of disk space. You would expect this information to be confirmed by entries in the root directory. Looking at Figure F-13, you can see that the entries for the first two files are as shown in Table F-2.

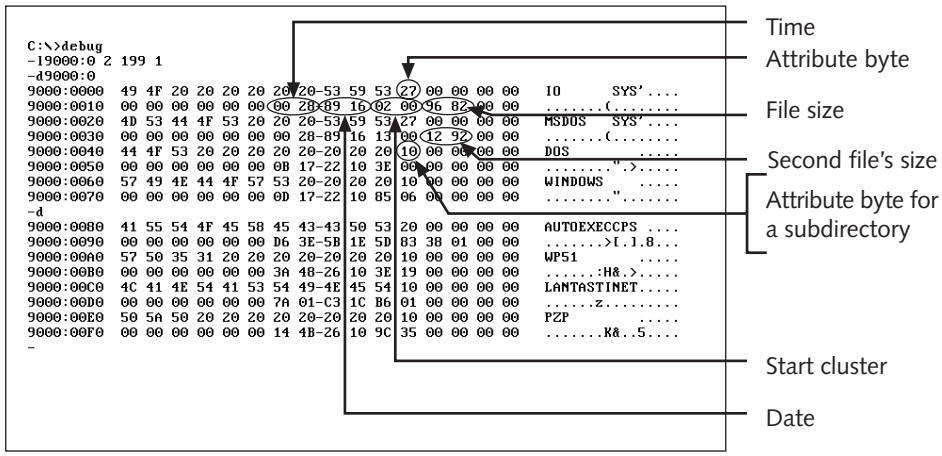

Figure F-13 A root directory

Table F-2 Example of directory entries for first two files in root directory

	File 1	File 2
Filename	IO	MSDOS
File extension	SYS	SYS
Attribute byte	27 = 0010 0111	27 = 0010 0111
Time	00 28	00 28
Date	89 16	89 16
Starting cluster number	00 02	00 13
Size of file	96 82	12 92

The hard drive is loaded with DOS and Windows 3.1. Not surprisingly, then, the first two files stored on this disk are the two DOS hidden files used to boot DOS. The attribute byte is interpreted in Table 5-5 of Chapter 5. From this Table 5-5 you can, therefore, deduce that the two files are system files that are not to be archived, and that they are hidden, read-only files.

According to what you saw in the FAT, the first file begins at cluster 2, and its size is written as 96 82. First reverse the bytes and then convert the hex number to decimal. The calculations for the hex number 8,296 are as follows:

8 × 4,096 = 32,768
2 × 256 = 512
9 × 16 = 144
6 × 1 = 6
TOTAL: 33,430

You saw from the FAT that this file occupies 17 clusters × 2,048 bytes/cluster, or 34,816 bytes of disk space. The unused part of the last cluster is the wasted space. The second file shows a size of 9,212h bytes, which is 37,394 in decimal. The file requires 19 clusters, or 38,912 bytes of space.

F

Some entries in the root directory are subdirectories rather than files. The third entry in Figure F-13 is the \DOS subdirectory. Note that the attribute byte for this entry is 10h, or 0001 0000 in binary. Table 5-5 of Chapter 5 shows that having the fourth bit on and all other bits off indicates a subdirectory entry. The starting cluster number for this entry is 00 3E, which is the beginning of the directory table for the \DOS subdirectory. It would be interesting to dump this sector of the hard drive and read this directory. Also note that the size of the file is zero for a subdirectory entry because DOS does not use this information.

You have seen that the root directory begins at logical sector 409. Where does it end, and where do data files begin? To calculate this, you need to know that there is room for 512 entries, or blocks, and that each entry is 32 bytes long. The root directory, then, occupies 512 entries × 32 bytes/entry = 16,384 bytes. To convert bytes to sectors, you divide 16,384 by 512 and see that the root directory is exactly 32 sectors long. Adding that to the starting sector 409, you find that the root directory ends at the 440th sector, and the data files begin at sector 441. In the FAT, sector 441 is named cluster 2. Therefore, to read a cluster number in the FAT table and to convert this number to a logical sector number on the hard drive you must use the following conversion expression, where 4 is the number of sectors in one cluster:

(cluster number – 2) × 4 + 441 = sector number

Examining the information at the beginning of a hard drive can be tedious, but knowledge is power. When you have to recover data from a damaged hard drive, skills learned here will make your work much easier.

Subdirectories

In Figure F-14 you see the DEBUG dump of the \GAME directory table. Note that the . and .. entries look just like file and subdirectory entries. Both show 10 or 0001 0000 as their attribute byte, indicating they are subdirectory entries rather than files. The starting cluster number of the . entry is 00 41, which is equal to 65 decimal. This byte marks the beginning of this directory table. The same 00 41 cluster number is found in the root directory as the starting cluster number of the \GAME directory. To use the DEBUG command to create the screen in Figure F-14, you need to locate this directory table. Convert the cluster number in decimal to the logical sector number according to the formula explained earlier:

(65 – 2) × 4 + 441 = 693 (decimal) = 2B5 (hex)

The DEBUG commands used to view the subdirectory table in Figure F-14 then look like this:

```
—L9000:0 2 2B5 1
—D9000:0
```

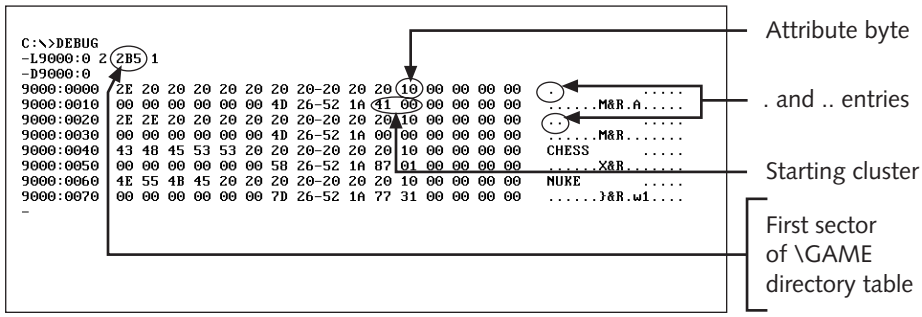

Figure F-14 Dump of subdirectory table C:\GAME

F

Directory Repair

This section gives you an in-depth look at the entries in a directory, to help you understand what happens when a utility like Norton Utilities repairs a directory. You can also use this information to help in the recovery of a corrupted file using Norton Disk Editor.

On the 3½-inch high-density disk used in the example below, the directory begins at sector 13h. Remember that the second copy of the FAT began at sector A and was nine sectors long. Thus, it ends at sector A + 8 or 12 in hex (10 + 8 = 18 in decimal), and the root directory begins with sector 13h. Figure F-15 is a memory dump of the root directory of the disk with the two files, MYFILE.TXT and Long Word File.doc. The root directory was first loaded into memory using this DEBUG command:

```
-L5000:0 0 13 1
```

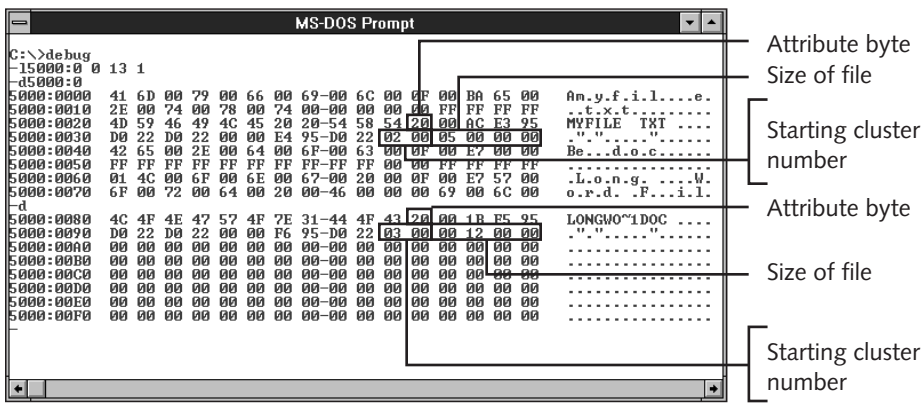

Figure F-15 Memory dump of a root directory containing two files

The layout of each entry in a directory for a file is listed in Tables 5-4 and 5-5 in Chapter 5. The first 32 bytes of the root directory contain the volume label of the disk. Each entry after that is also 32 bytes long for DOS and longer for Windows 9x when using long filenames. The extra information for the long filename is listed in the root directory before the 32 bytes used by both DOS and Windows 9x for each file entry.

Look at the entries for MYFILE.TXT and Long Word File.doc in Figure F-15. The attribute byte, starting cluster number, and size of the file are labeled for each of the two files listed in this root directory. The second file has a long filename together with its short DOS filename. The size of the file Long Word File.doc is stored in 4 bytes at the end of the 32-bit entry of the file that follows the long filename information (see addresses 5000:0080 of the dump). These four bytes read:

```
00 12 00 00
```

To interpret them, you must read in reverse order like this:

```
00 00 12 00
```

You can then convert the hex 1200 to decimal, which is 4,608, the size of the file.

In Figure F-15, the attribute byte for this file is at memory location 5000:008B. The byte reads 20 in hex. Converting 20h to binary so that you can read each bit, you take each hex number and convert it to binary like this:

2 in hex = 0010 in binary, and 0 in hex = 0000 in binary, therefore the byte is 0010 0000

Looking back at Table 5-5 in Chapter 5, you can interpret each of these 8 bits, reading from left to right, in the following way:

- 0 Not used
- 0 Not used
- 1 To be archived
- 0 A file, not a directory
- 0 A normal directory entry, not a volume label
- 0 A normal file, not a system file (like IO.SYS)
- 0 Not hidden
- 0 Read/write file, not read-only

By understanding how to interpret each item in the directory entry, you can reconstruct a damaged directory table if you need to. Again, you may never use DEBUG to repair a directory because utility software is so much easier to use, but the purpose of using it here is to learn exactly how entries are written to the directory, so you can understand what these utilities are doing for you. Norton Utilities Disk Editor makes reading and writing to directories very easy to do. This editor converts the entries to decimal for display, accepts your decimal entry, and converts it to hex and binary before writing it to the disk.